Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Homework 10 due on 4/4
  - Lab 10 due on 4/8
  - Homework 11 due on 4/11
  - Assignment 3 and 4 due on 4/18
    - Used to be one assignment

Previous lecture …

- Weighted Shortest Path
  - Dijkstra’s Single-source shortest-paths algorithm

CourseMIRROR Reflections (most confusing)

- What exactly was meant by tentative distance
- The Dijkstra algorithm was a bit confusing in regard to what order to visit each node
- When do we need to check for a disconnected graph for Dijkstras Algorithm?
- Why would dijkstra get his own algorithm named after him if it’s the same as prim’s? Or vice versa? Seems a bit trivial
  - Dijkstra’s: conceived 1956, published 1959
  - Prim’s: discovered in 1930 by Jarník, published 1957 and 1959, also called Dijkstra-Jarník-Prim (DJP)
- how does dijkstra’s differ from prim’s?
- Forest of trees implementation from last lecture is still a little confusing
- why is find bound by the height of the tree?
- Why does introducing a weighted tree make the runtimes LogN for union/find
- I didn’t really follow path compression at all besides that it is constant time, how does path compression work?
- The runtime analysis table

CourseMIRROR Reflections (most interesting)

- Differences and similarities between dijkstras and Kruskals and between dijkstras algorithm and prims
- Dijkstras algorithm, it’s clear and simple and useful.
- The last example relates Dijkstra
- algorithm optimizations
- Working through dynamic connectivity problem and seeing difference in representation between array and tree
- I found the path compression optimization to be interesting for Union/Find
- The Union/Find ADT implementations were most interesting.
- How to reduce the runtime of find and union both to log(n)
- going over heap sorting and prim’s examples helped a lot

Problem of the Day: Finding Bottlenecks

- Let's assume that we want to send a large file from point A to point B over a computer network as fast as possible over multiple network links if needed
- Input:
  - A computer network
    - Network nodes and links
    - Links are labeled by link capacity in Mbps
  - Starting node and destination node
- Output:
  - The maximum network speed possible for sending a file from source to destination

Defining network flow

- Consider a directed, weighted graph G(V, E)
  - Weights are applied to edges to state their *capacity*
    - c(u, w) is the capacity of edge (u, w)
    - if there is no edge from u to w, c(u, w) = 0
- Consider two vertices, a *source s* and a *sink t*
  - Let's determine the maximum flow that can run from s to t in the graph G

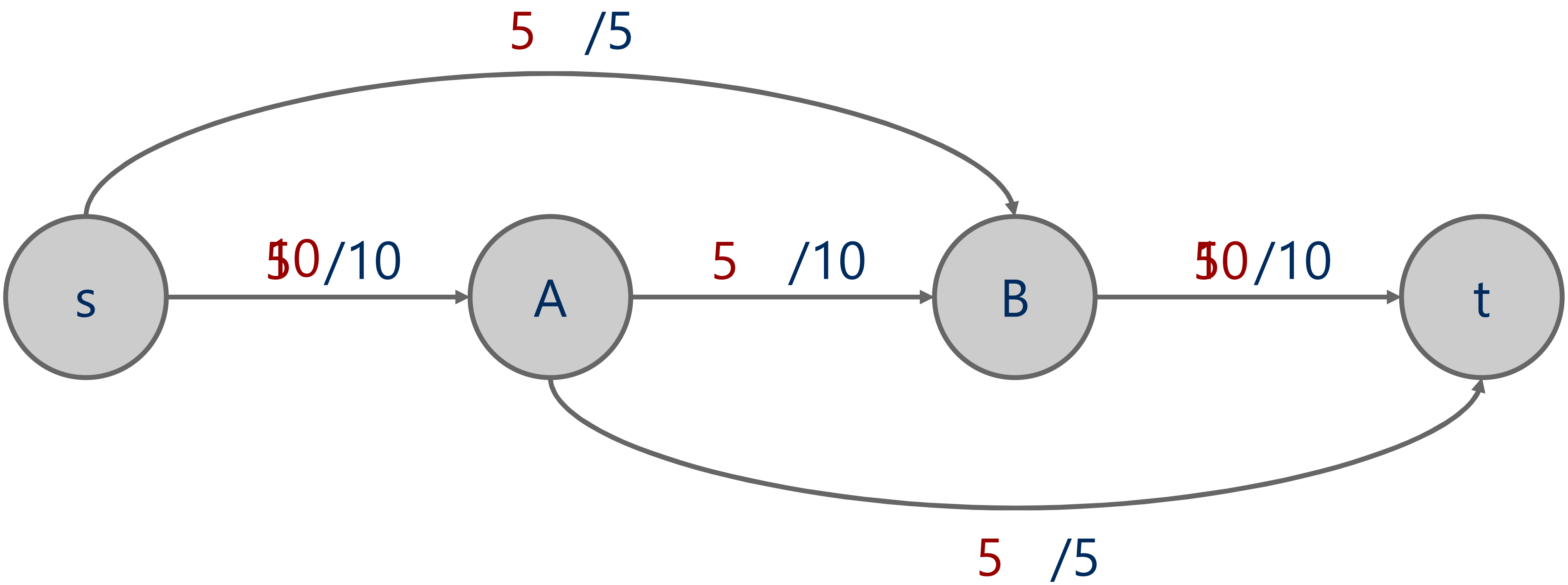
Flow

- Let the f(u, w) be the amount of flow being carried along the edge (u, w)
- Some rules on the flow running through an edge:
  - $\forall (u, w) \in E \; f(u, w) <= c(u, w)$
  - $\forall u \in (V - \{s,t\}) \; (\Sigma_{w \in V} f(w, u) - \Sigma_{w \in V} f(u, w)) = 0$

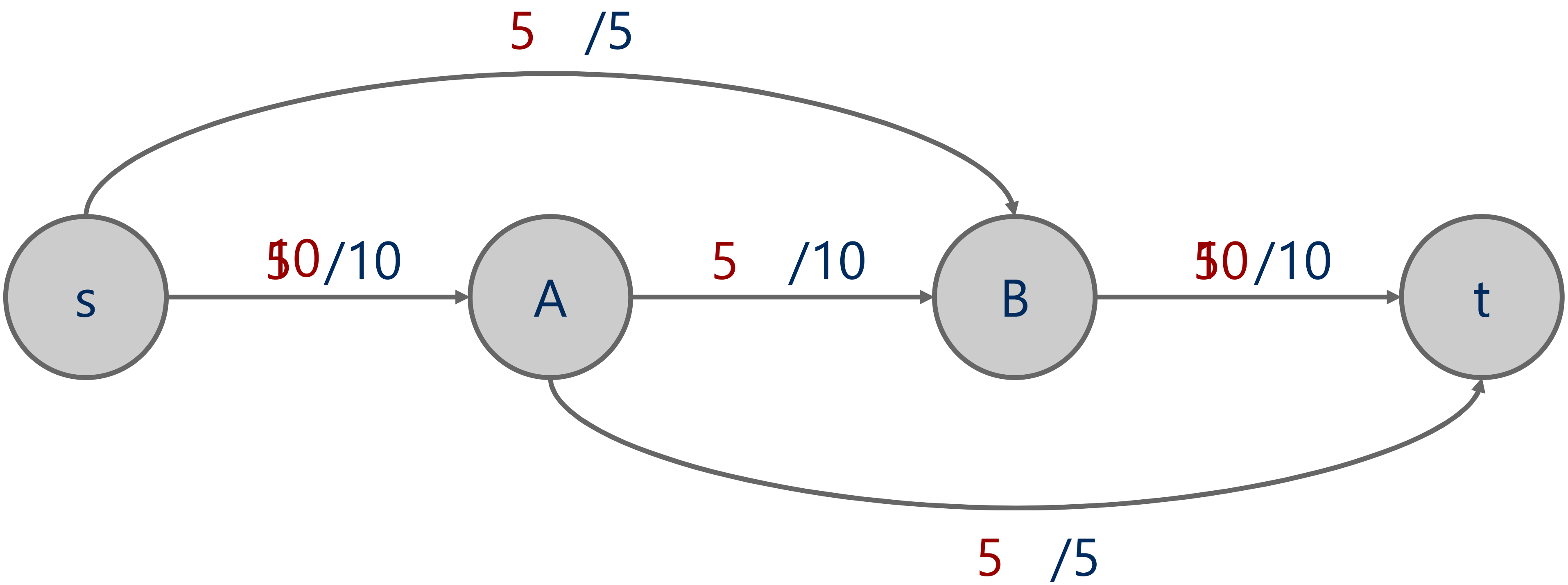
Ford Fulkerson

- Let all edges in G have an allocated flow of 0
- While there is path *p* from s to t in G s.t. all edges in p have some *residual capacity* (i.e., $\forall(u, w) \in p\ f(u, w) < c(u, w)$):
  - (Such a path is called an *augmenting path*)
  - Compute the residual capacity of each edge in p
    - Residual capacity of edge (u, w) is c(u, w) - f(u, w)
  - Find the edge with the minimum residual capacity in p
    - We'll call this residual capacity *new_flow*
  - Increment the flow on all edges in p by new_flow

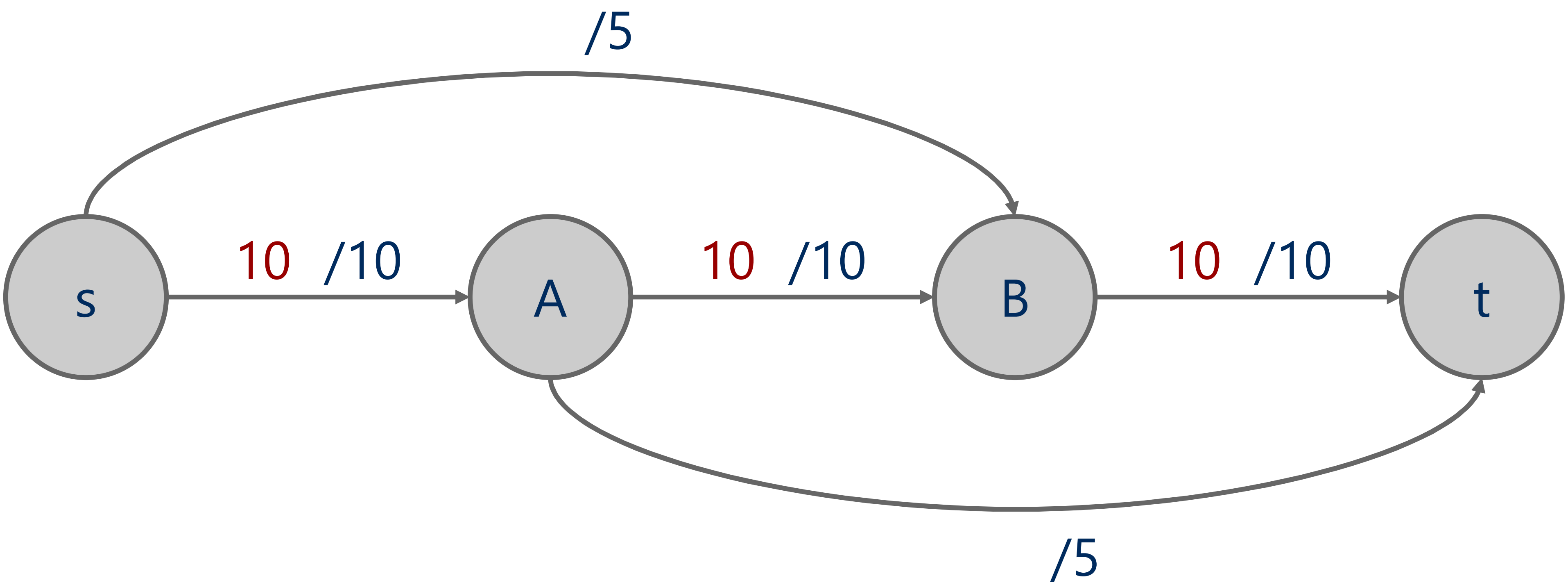
Ford Fulkerson example



Ford Fulkerson example



Another Ford Fulkerson example



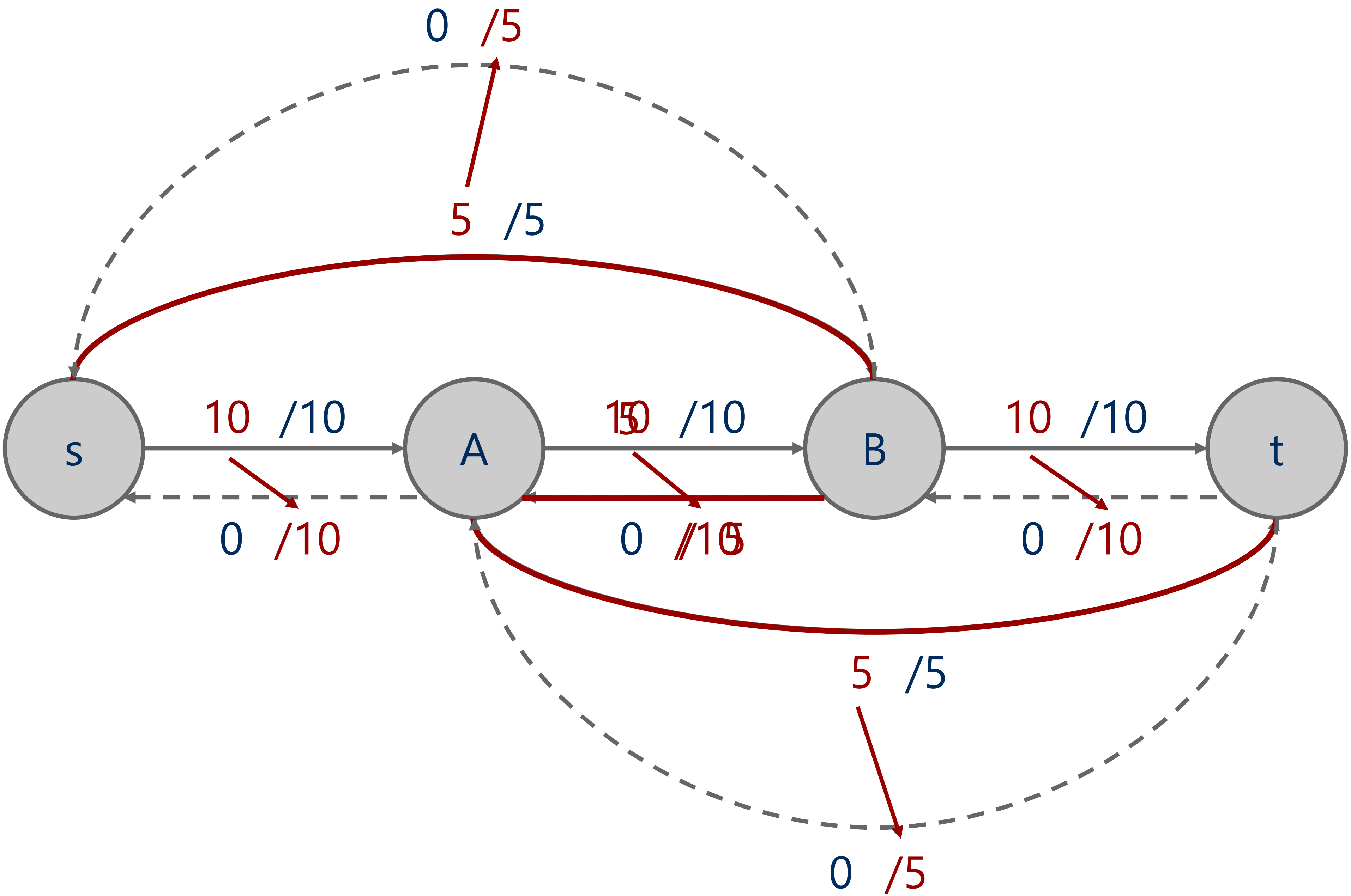
Expanding on residual capacity

- To find the max flow we will have to consider re-routing flow we had previously allocated
  - This means, when finding an augmenting path, we will need to look not only at the edges of G, but also at *backwards edges* that allow such re-routing
    - For each edge $(u, w) \in E$, a backwards edge (w, u) must be considered during pathfinding if $f(u, w) > 0$
      - The capacity of a backwards edge (w, u) is equal to $f(u, w)$

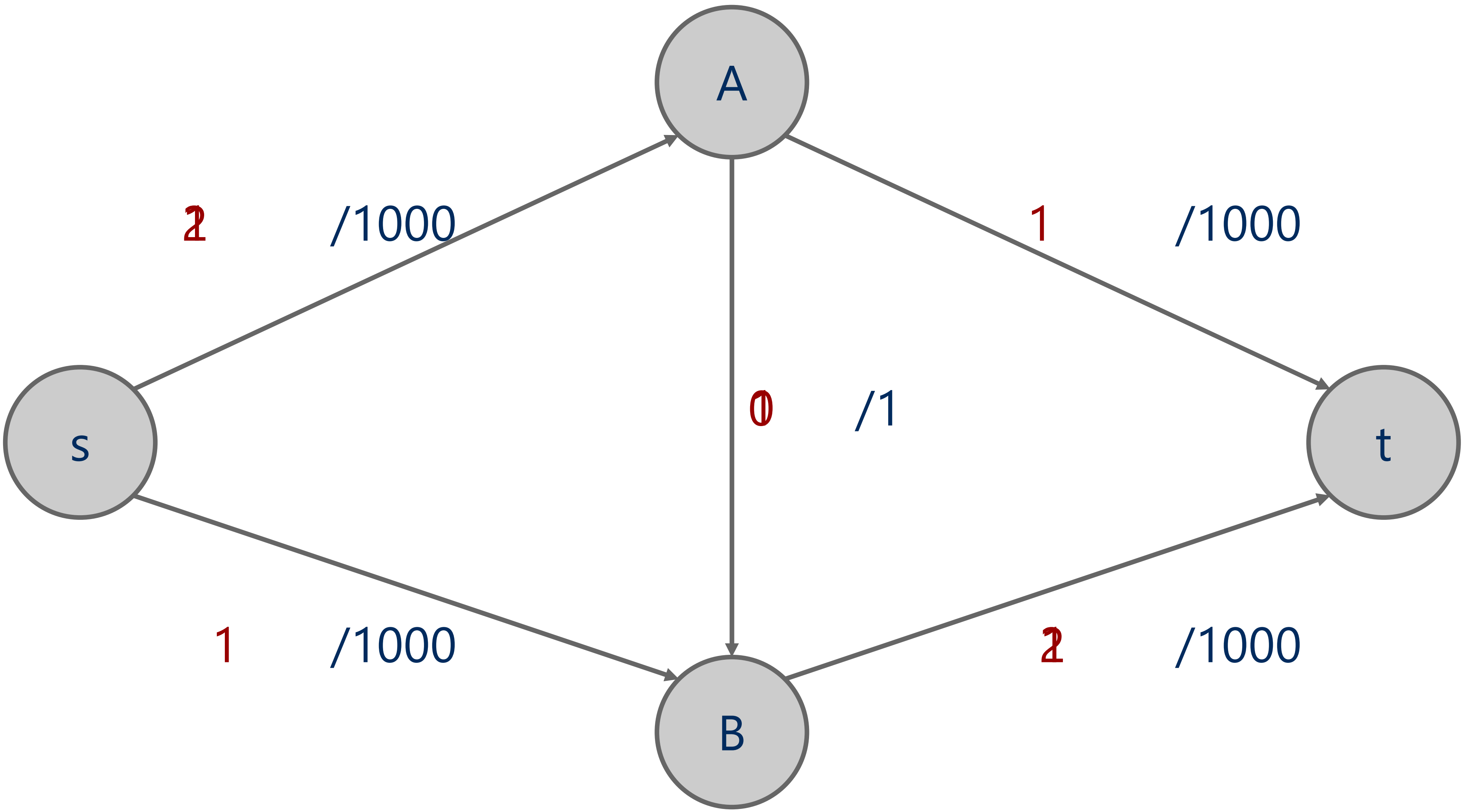
The residual graph

- We will perform searches for an augmenting path not on G, but on a residual graph built using the current state of flow allocation on G
- The residual graph is made up of:
  - V
  - An edge for each $(u, w) \in E$ where $f(u, w) < c(u, w)$
    - (u, w)'s mirror in the residual graph will have 0 flow and a capacity of $c(u, w) - f(u, w)$
  - A backwards edge for each $(u, w) \in E$ where $f(u, w) > 0$
    - (u, w)'s backwards edge has a capacity of $f(u, w)$
    - All backwards edges have 0 flow

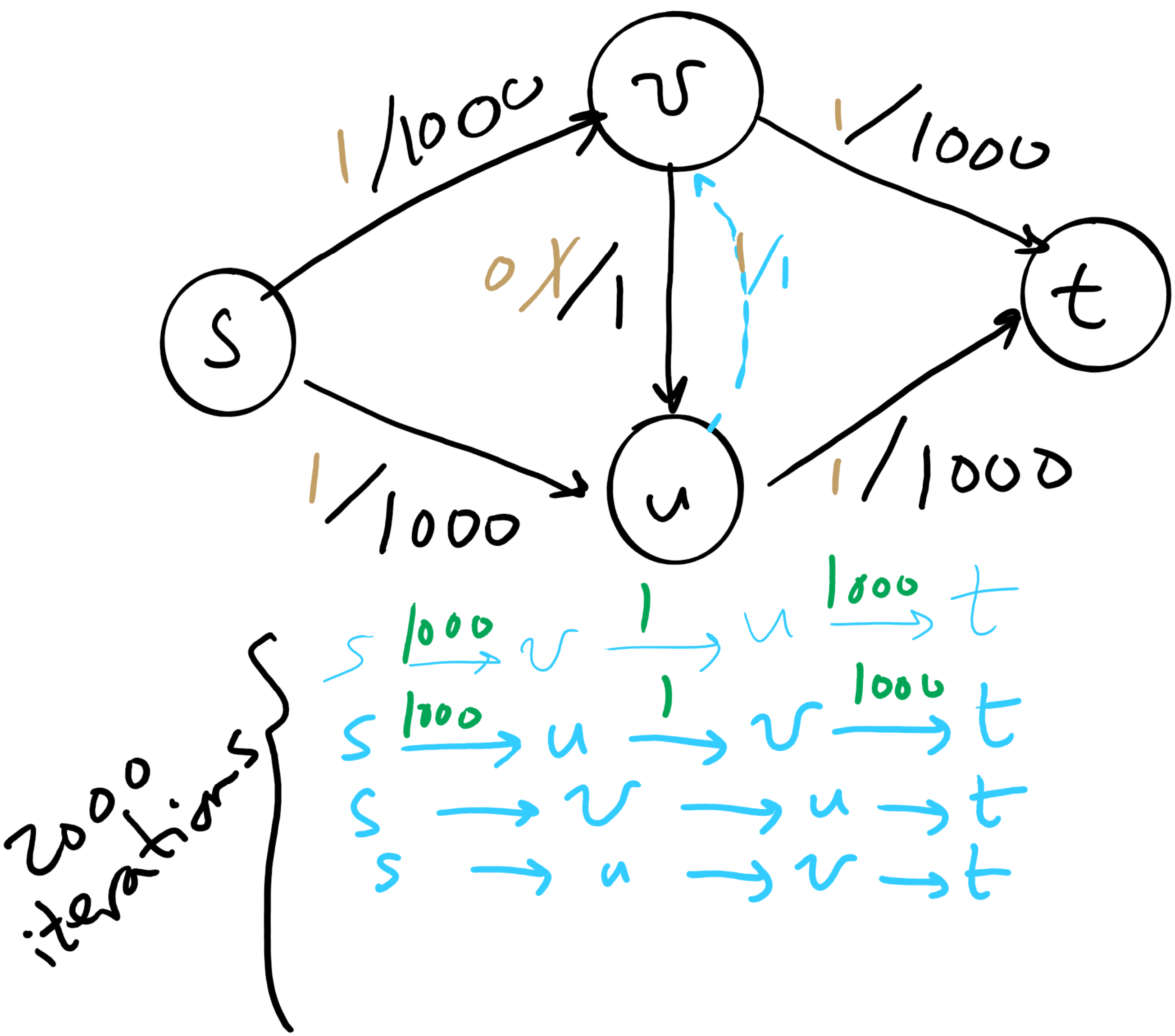
Residual graph example



Another example



Worst-case runtime of Ford-Fulkerson



Worst-case Runtime of Ford-Fulkerson

$$\Theta\left(|f| * (e+v)\right)$$

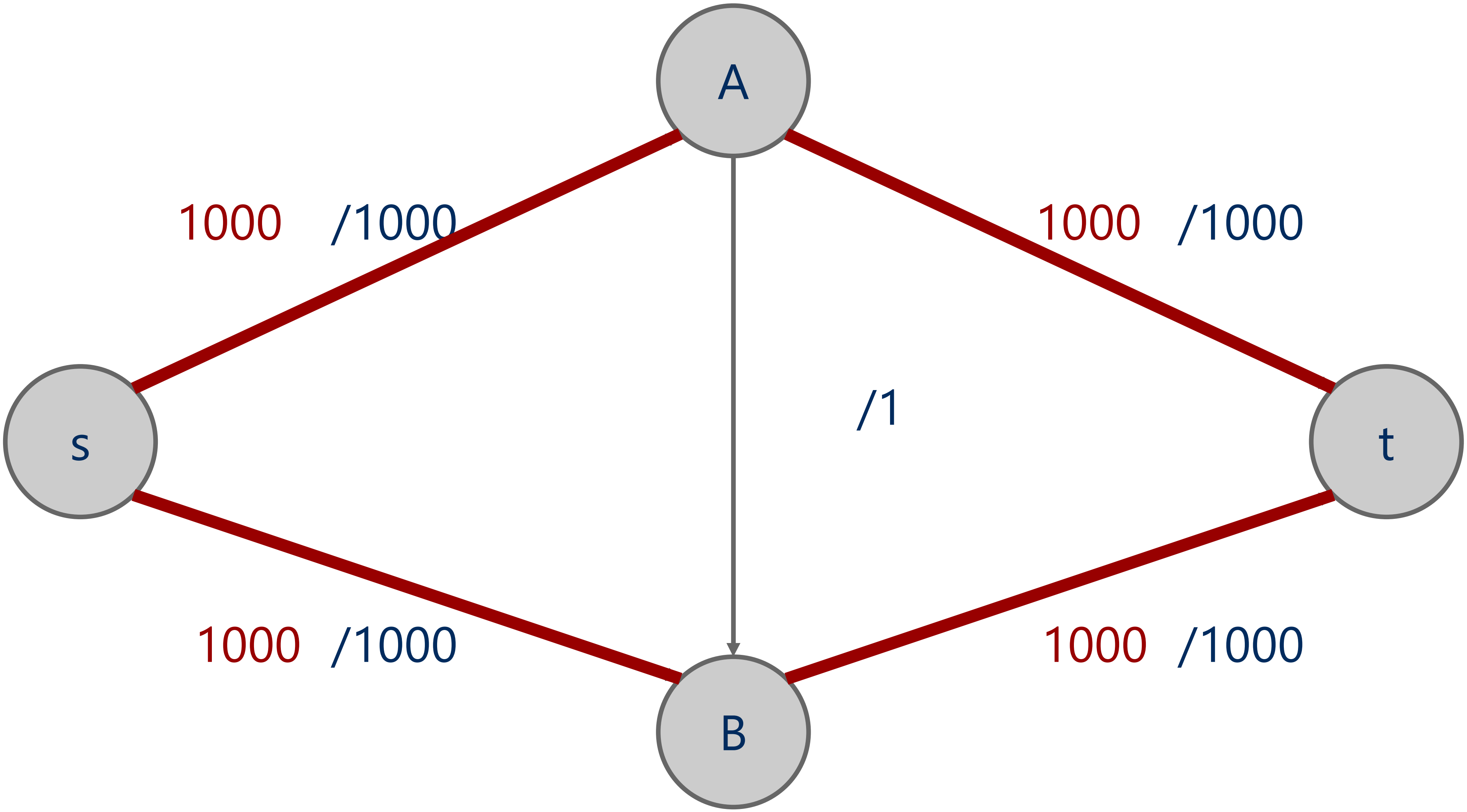
Max. Flow Value

time for finding an augmenting path

Edmonds Karp

- How the augmenting path is chosen affects the performance of the search for max flow
- Edmonds and Karp proposed a shortest path heuristic for Ford Fulkerson
  - Use BFS to find augmenting paths

Another example

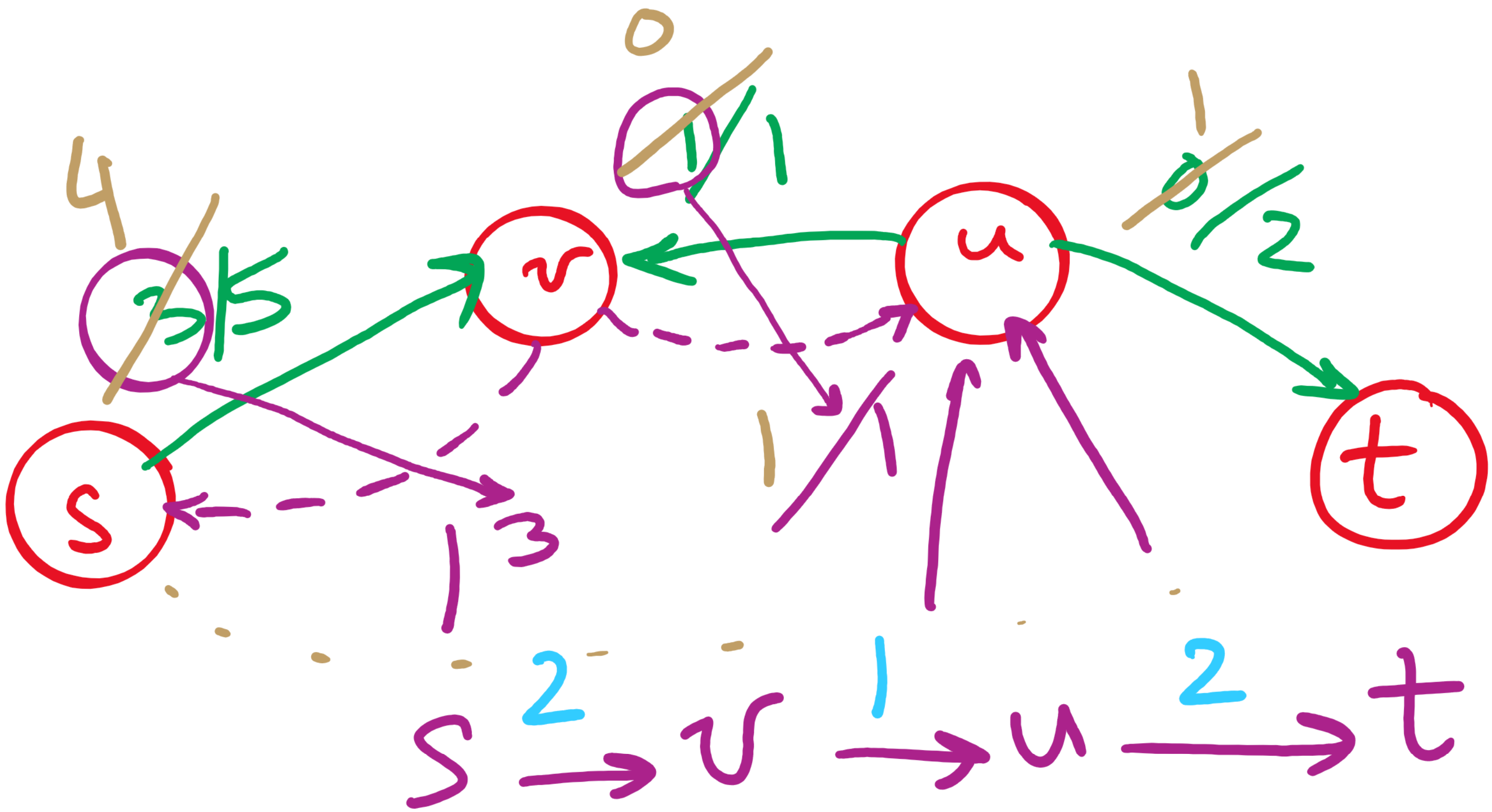


But our flow graph is weighted...

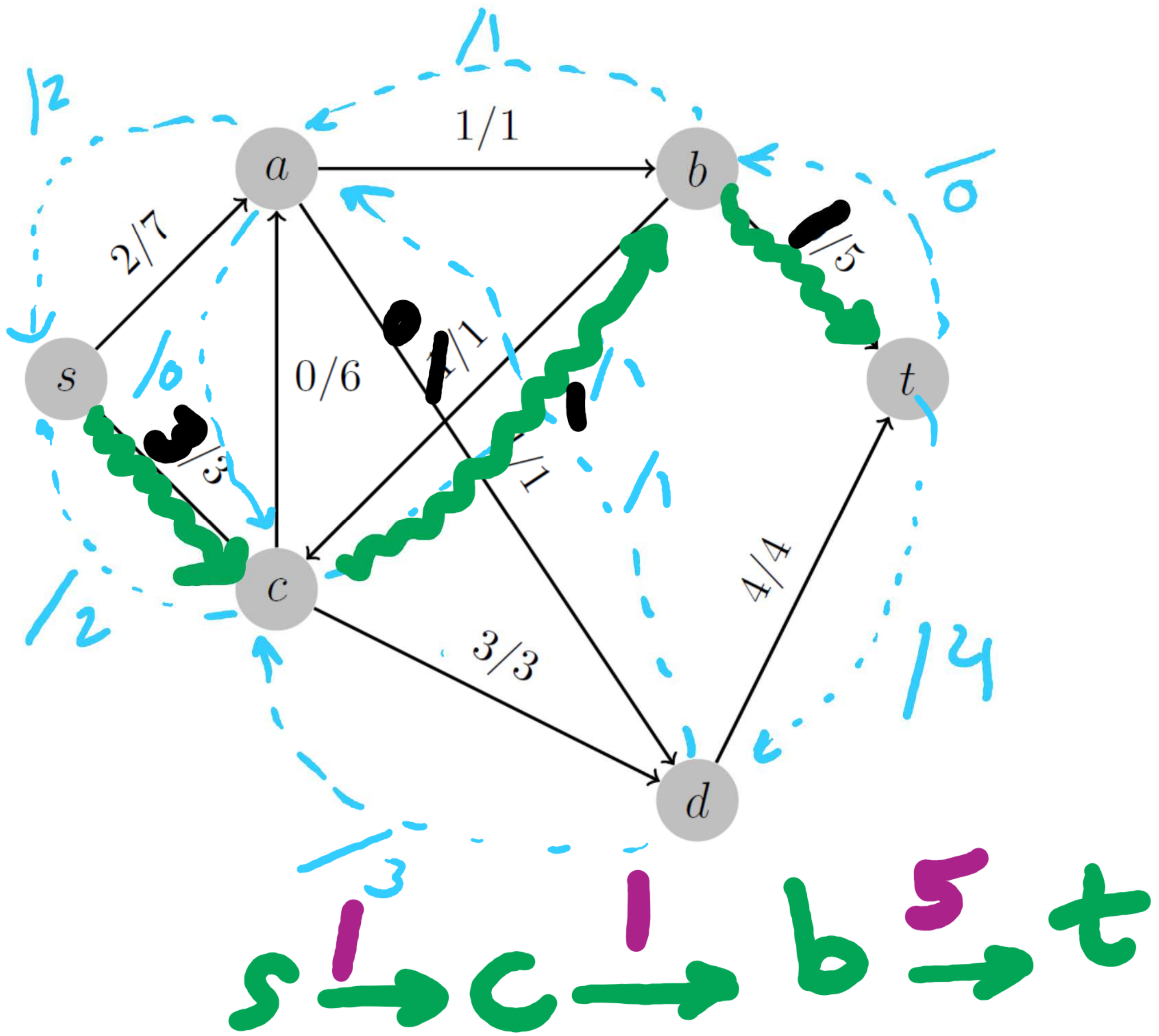
- Edmonds-Karp only uses BFS
  - Used to find spanning trees and shortest paths for *unweighted* graphs
  - Why do we not use some measure of priority to find augmenting paths?

Backwards edges

- Adding flow to a backwards edge means rerouting flow from the corresponding forward edge



2nd Tophat Question



Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com